Alfio Lazzaro: "Introduction to OpenMP"

- **Section I**
  - Basic ideas and syntax concepts
  - Getting started with the very first program
- **Section II**
  - Getting parallelism and speed-up
  - What can make life difficult: conflicts
- **Section III**
  - Some OpenMP clauses: reductions, critical sections, single sections
  - Synchronization controls

Based on Sverre Jarp/CERN Openlab @ CERN Openlab Multi-Threading and Parallelism Workshop (Material originally from Hans-Joachim Plum, Intel GmbH)

References

- Books:
  - "Using OpenMP: Portable Shared Memory Parallel Programming", Chapman, Jost, van der Pas, http://www.amazon.com/Using-OpenMP-Programming-Engineering-Computation/dp/0262533022/

- Online tutorials:
  - https://computing.llnl.gov/tutorials/openMP/
- Reference page:
  - http://www.openmp.org

Section I

- Basic ideas and syntax concepts
- Getting started with the very first program

What is OpenMP

- Compiler directives and library calls for multi-threaded programming
  - Easy to create threaded C/C++ and Fortran codes
  - Explicit parallelization
    - Especially oriented for loop parallelization
  - Supports the data parallelism model for shared memory paradigm
  - Offers incremental parallelism
  - Combines serial and parallel code in a single source

What is OpenMP

```
C$OMP FLUSH
#pragma omp critical
C$OMP THREADPRIVATE(/ABC/)
CALL OMP_SET_NUM_THREADS(10)
C$OMP parallel do shared(a, b, c)
call omp_test_lock(jlok)
call OMP_INI
C$OMP MASTER
C$OMP SINGLE PRI
C
ynamic”
C$OMP PARALLEL
C$OMP ORDERED
C$OMP PARALLEL
IONS
#pragma omp parallel for private(A, B)
!$OMP BARRIER
C$OMP PARALLEL COPYIN(/blk/)
C$OMP DO lastprivate(XX)
Nthrds = OMP_GET_NUM_PROCS()
omp_set_lock(lck)
```

http://www.openmp.org

Current spec is OpenMP 3.0

326 Pages

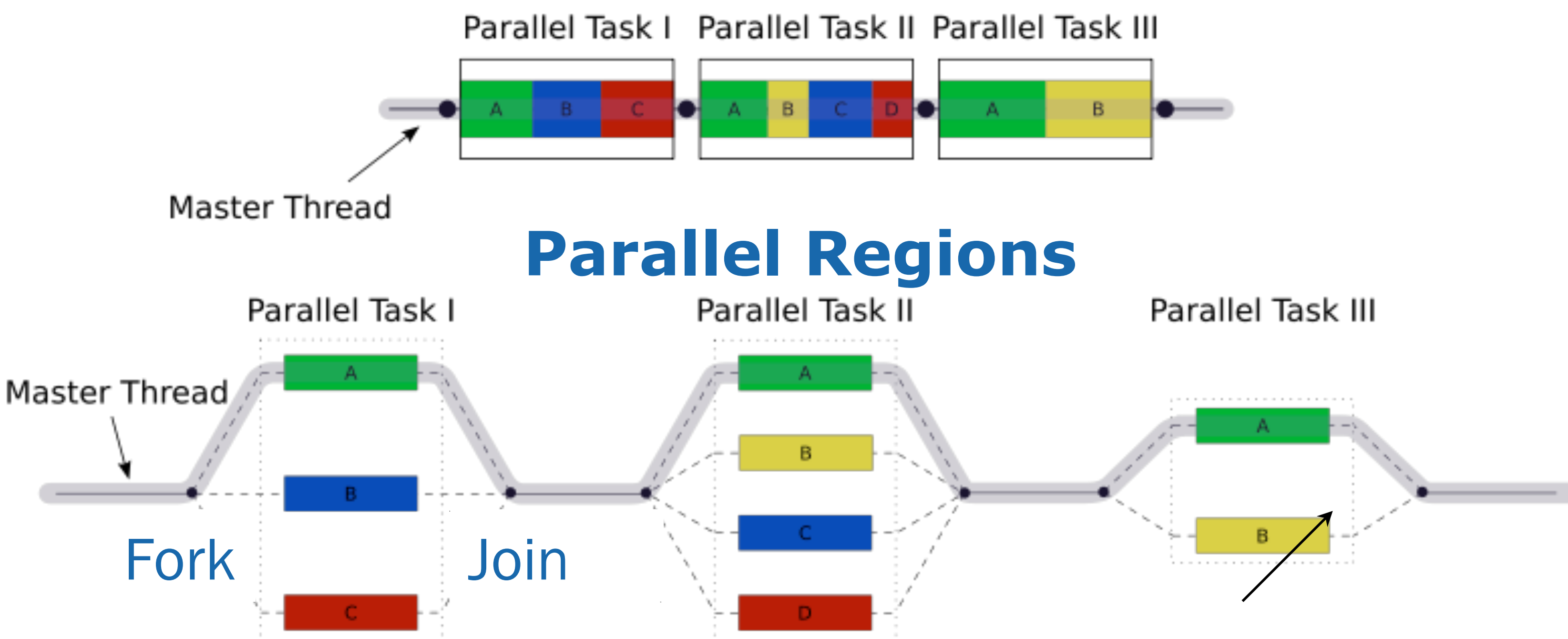
(combined C/C++ and Fortran)

Terminology

- Variables can be
  - Private: Visible to one thread only
    - Change made in local data, is not seen by others
    - Example: Local variables in a function that is executed in parallel
  - Shared: Visible to all threads
    - Change made in global data, is seen by all others
    - Example: Global data
- OpenMP team: Master + Workers
  - The master thread always has thread ID 0
- A parallel region is a block of code executed by all threads simultaneously
- A work-sharing construct divides the execution of the enclosed code region among the members of the team

Programming Model

- Fork-Join parallelism:
  - Master thread spawns a team of threads as needed
  - Parallelism is added incrementally: the sequential program evolves into a parallel program



OpenMP pragma syntax

- Most constructs in OpenMP are compiler directives or pragmas
  - For C and C++, the pragmas take the form:

```
#pragma omp construct [clause [clause]…]
```

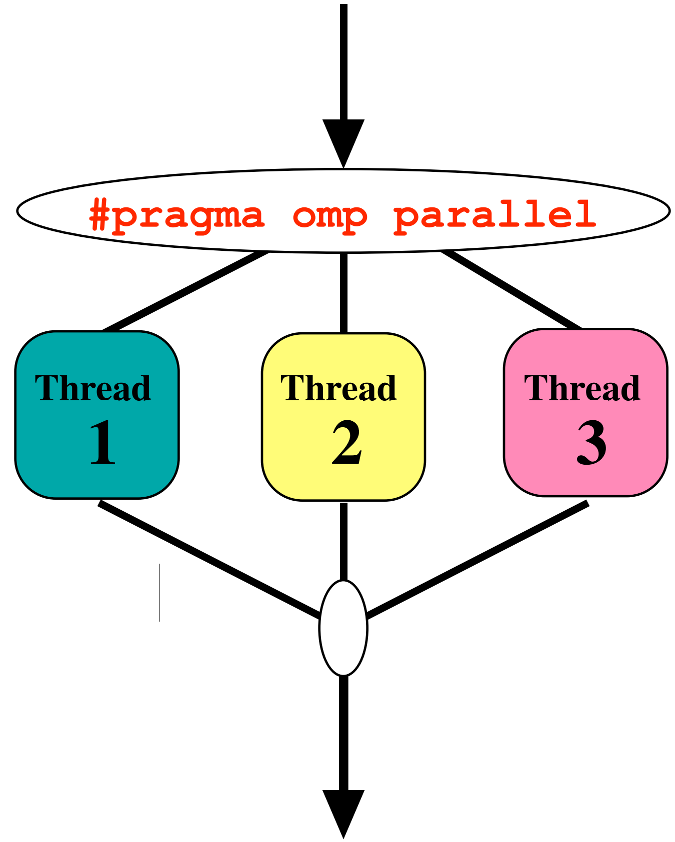
  - For example:

```
#pragma omp parallel for private(A, B)
```

I will use C++ in this lecture

Parallel Regions

- Threads are created as "**parallel**" when the pragma is crossed
- Threads block at end of region
- Data is shared among threads unless specified otherwise
- Parallel regions can be nested, but support for this is implementation dependent
- An `if` clause can be used to guard the parallel region; in case the condition evaluates to "false", the code is executed serially



C/C++ :

```
#pragma omp parallel
   {
         block
   }
```

Defining number of threads

- Set environment variable for number of threads:

```
export OMP_NUM_THREADS=4
```

- There is no standard default for this variable
  - Many systems:
    - # of threads = # of CPUs as in "cat /proc/cpuinfo"
    - Intel compilers use this default

Getting Started: Hello World

```
#include <omp.h> // only in case you use openmp functions
#include <iostream>

int main() {
  // Default, normal serial execution:
  std::cout << "Program running before parallel region" << std::endl;
  int diagnostics = 7777;

#pragma omp parallel
  // Now, the following block is executed by multiple threads:
{
  std::cout << "Thread " << omp_get_thread_num()
            << " / " << omp_get_num_threads() << ": "
            << "Thread running in parallel region " << diagnostics
            << std::endl;
}
  // end omp parallel

  // Back to normal serial execution:
  std::cout << "Program ending after parallel region" << std::endl;
}
```

Compile and Run

- Compilation
  - Intel

```
-bash-3.00$ icpc -openmp helloworld.cxx -o helloworld
helloworld.cxx(9) : (col. 1) remark: OpenMP DEFINED
REGION WAS PARALLELIZED.
```

  - GNU (since version 4.2)

```
-bash-3.00$ g++ -fopenmp helloworld.cxx -o helloworld
```

- Decide #threads to run and run

```
-bash-3.00$ export OMP_NUM_THREADS=3
-bash-3.00$ ./helloworld
Program running before parallel region
Thread 0 / 3: Thread running in parallel region 7777
Thread 1 / 3: Thread running in parallel region 7777
Thread 2 / 3: Thread running in parallel region 7777
Program ending after parallel region
```

Section II

- Getting parallelism and speed-up
- What can make life difficult: conflicts

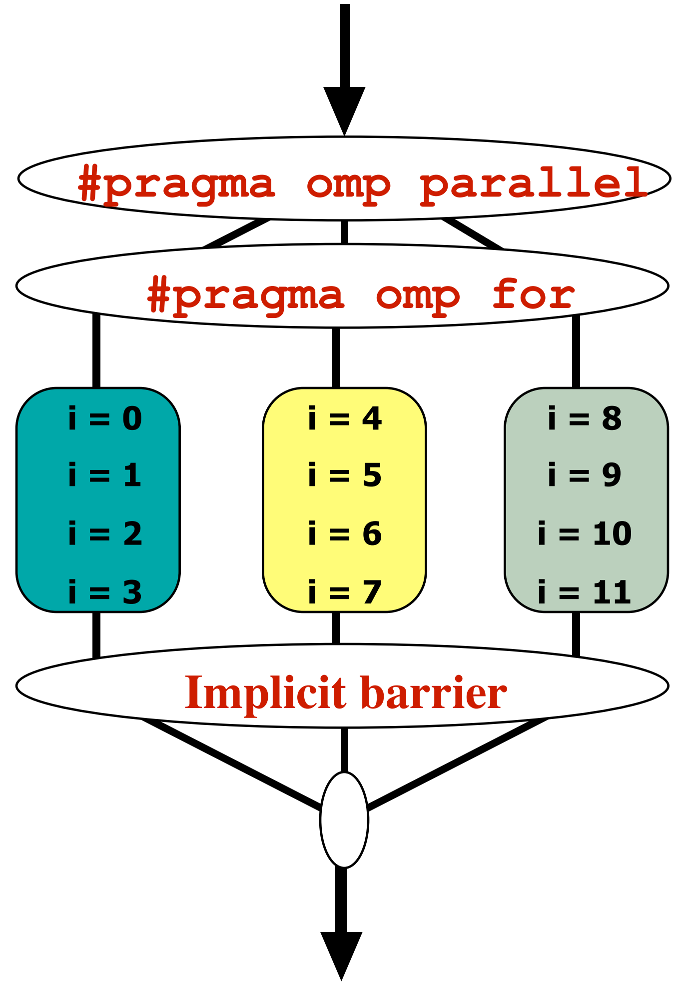
Making it work in parallel

- Work-sharing construct:
  - used to specify how to assign independent work to one or all of the threads
  - Must be enclosed in a parallel region
- Case of loops
  - Splits loop iterations into threads
  - Must precede the loop

```
#pragma omp parallel
{
  #pragma omp for
  for (i=0; i<N; i++) {
      Do_Work(i);
  }
}
```

Work-sharing construct

- Each thread is assigned an independent set of iterations
- Threads must wait at the end of the work-sharing construct
- Few restrictions:
  - *loop_variable* must be signed integer
  - not possible to use break to go out from the loop
  - Comparison in the form *loop_variable* <, <=, >, or >= *loop_invariant_integer*
    - *loop_variable* must increment (decrement) on every iteration if the condition is < or <= (> or >=)
  - The increment portion must be either integer addition or integer subtraction and by a loop invariant value




```
#pragma omp parallel
#pragma omp for
   for(i = 0; i < 12; i++)
      c[i] = a[i] + b[i]
```

Example: An "easy loop"

```
const int N = 500000;
double a[N], b[N];
// initialization of the vectors; skip

double stime = omp_get_wtime(); // start timer

#pragma omp parallel
// Now, the following block is executed by multiple threads:
{
 #pragma omp for
 for (int i = 0; i < N; i++) {
   a[i] = exp(a[i])/exp(b[i]);
   b[i] = 0.111+exp(a[i])+exp(b[i]);
   // other operations...
 }
}

double etime = omp_get_wtime(); // end timer

std::cout << "Time is " << (etime-stime)*1e3 << " milliseconds."
          << std::endl;
```

Compile and run "the easy loop"

```
icpc -openmp easy_loop.cxx -o easy_loop
easy_loop.cxx(17) : (col. 1) remark:
OpenMP DEFINED LOOP WAS PARALLELIZED.
easy_loop.cxx(14) : (col. 1) remark:
OpenMP DEFINED REGION WAS PARALLELIZED.
```

```
-bash-3.00$ export OMP_NUM_THREADS=4
-bash-3.00$ ./easy_loop
Time is 95.268 milliseconds.
-bash-3.00$ export OMP_NUM_THREADS=1
-bash-3.00$ ./easy_loop
Time is 331.269 milliseconds.
```

=> speedup ~3.5

An example that goes wrong!

```
double x, y;
int i;

#pragma omp parallel
{
#pragma omp for
  for(i=0; i<N; i++) {
    x = a[i]*a[i];
    y = b[i]*b[i];

    b[i] = x + y + x*y;
  }
}
```

- Why?
- Who can explain?

Needed: the **private** clause

x and y cannot be shared!

```
double x, y;
int i;

#pragma omp parallel
{
#pragma omp for private(x,y)
  for(i=0; i<N; i++) {
    x = a[i]*a[i];
    y = b[i]*b[i];

    b[i] = x + y + x*y;
  }
}
```

The `private` clause

- Make a local copy of the variables for each thread and use them as temporary variables
  - Variables not initialized; C++ object is default constructed
  - the values are not maintained for use outside the parallel region, i.e. any value external to the parallel region is undefined
- What about the loop variable `i`?
  - By default, the loop variables in the OpenMP loop constructs are automatically private

Data Environment

- Most variables are shared by default (shared-memory programming model)
  - Global variables are shared among threads
    - C/C++: File scope variables, static
- But, in some cases, private is the default:
  - Stack variables in functions called from parallel regions
  - Loop index variables (with some exceptions)

Section III

- Some OpenMP clauses: reductions, critical sections, single sections
- Synchronization controls

Sums and Reductions

```
float dot_prod(float* a, float* b, int N) {
  float sum = 0.0;
  #pragma omp parallel
  #pragma omp for
  for(int i = 0; i<N; i++) {
     sum += a[i] * b[i];
  }
  return sum;
}
```

- Code is wrong due to conflicts on **`sum`**
- However, **`sum`** is not private, but a global so-called reduction variable

Sums and Reductions

- OpenMP provides a clause for such variables:

```
float dot_prod(float* a, float* b, int N) {
  float sum = 0.0;
  #pragma omp parallel
  #pragma omp for reduction(+: sum)
  for(int i=0; i<N; i++) {
     sum += a[i] * b[i];
  }
  return sum;
}
```

- Implicitly, there is a local copy of `sum` for each thread
  - At the end, all local copies are added together and stored in the original variable

OpenMP **reduction** clause

```
reduction (op : var_list)
```

- The variables in `var_list` must be shared in the enclosing parallel region
- Inside work-sharing construct:
  - A private copy of each list variable is created and initialized depending on the `op`
  - These copies are updated locally by threads
  - At end of construct, local copies are combined through `op` into a single value and combined with the value in the original shared variable

C/C++ **`reduction`** operations

- A range of associative and commutative operators can be used with reduction
- Initial values are the ones that make sense

Operator	Initial Value
+	0
*	1
-	0
^	0

Operator	Initial Value
&	~0
|	0
&&	1
||	0

Control of mapping: Assigning Iterations

- Examples **static**:

```
#pragma omp for schedule (static, 8)
     for (int i = start; i <= end; i += 2)
     {         // work       }
```

  - Iterations are divided into chunks of 8
  - If **start = 3**, then first chunk is **i={3,5,7,9,11,13,15,17}**
  - Chunks are executed round-robin by parallel threads

Control of mapping: Assigning Iterations

- Examples **dynamic**:

```
#pragma omp for schedule (dynamic, 8)
     for (int i = start; i <= end; i += 2 )
     {         // work      }
```

  - As **static**, but chunks are always (dynamically) assigned to the next free thread; can be useful for uneven workloads

OpenMP **`critical`** Construct

- When certain pieces of a parallel region must be executed only one thread at a time

```
#pragma omp parallel
{
  #pragma omp for
  for (i_el=0; i_el<N_elements; i_el++) {

    // major piece of parallel work
    // involving the i_el element

    #pragma omp critical // One thread at a time
    {
      // minor piece of serial work
    }
  }
}
```

OpenMP **`single`** Construct

- When certain pieces of a parallel region must only be executed once; which thread does it, doesn't matter

```
#pragma omp parallel
{
  // do parallel work part 1

  #pragma omp single
  {
    // only first-come thread executes
  }

  // do parallel work part 2

}
```

Other OpenMP Constructs

- **`sections`**
  - distribute different independent code sections to threads (functional parallelism)
- **`master`**
  - as the **`single`** pragma, but by master thread
- **`ordered`**
  - As **`critical`**, but stricter: threads must execute serial and maintain the original order of the loop
- Advanced uses: **`atomic`** pragma and locks
  - manually synchronized concurrent updates of global variables

References

- Books:
  - "Using OpenMP: Portable Shared Memory Parallel Programming", Chapman, Jost, van der Pas, http://www.amazon.com/Using-OpenMP-Programming-Engineering-Computation/dp/0262533022/

- Online tutorials:
  - https://computing.llnl.gov/tutorials/openMP/
- Reference page:
  - http://www.openmp.org